L8 - Collections and Iterators

February 11, 2020

1 More collections

1.1 Searching

To search:

- inherently indefinite
- must handle success and failure
  - success: found a thing
  - failure: nowhere left to look
- `while` loop syntax means we must specify the conditions to *continue*
  - `searching == true && index < list.size()`
  - if we find a thing:
    - `searching = false`
- alternately:
  - `found == false && index < list.size()`
  - `found = true` if we find a thing
- if we do all that and reach the end of the loop, either we found the thing at some index or we got to the end of the collection without finding it

Note that a while loop can be executed **zero** times if the condition is never true.

[ ]:
```
import java.util.ArrayList;
ArrayList<String> files = new ArrayList<>();

files.add("LOOPSBROETHER.gif");
```

[ ]:
```
public String searchQuery = "deadmeme.gif";

int index = 0;
boolean found = false;

while ( index < files.size() && !found )
{
```

```
    String file = files.get(index);
    if ( file == searchQuery )
    {
        found = true;
    }
    else
    {
    index++;
    }
}

System.out.println(found);
```

1.1.1 An issue

The Java compiler merges identical `String` objects into a single reference to a single object. This is fine in many cases, but cannot be done with strings that arise from outside the program (like input data). Instead, use the `Strings.equals()` method:

- `if ( input == "bye" ) {}` - don't do this
  - `==` is the *identity* operator: the input is not *the same reference* as "bye"
  - if there's *anything* different about these objects, these won't be the same object
- `if ( input.equals("bye") ) {}` - do this
  - method compares the actual string payload

1.2 `while`without a collection

[ ]:
```
// even numbers from 2 to 30
int index = 2;
while (index <= 30)
{
    System.out.println(index);
    index = index + 2;
}
```

1.3 Not just strings

If we want to implement advanced functionality to the music organizer, we could instead implement a `Track` class which would implement useful functions like having fields for album, artist, year, label, etc. It could alse have a `playTrack` method which could return the filename for passing to the player.

1.3.1 Music Organizer v5

[ ]:
```
/**
 * Store the details of a music track,
 * such as the artist, title, and file name.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class Track
{
    // The artist.
    private String artist;
    // The track's title.
    private String title;
    // Where the track is stored.
    private String filename;

    /**
     * Constructor for objects of class Track.
     * @param artist The track's artist.
     * @param title The track's title.
     * @param filename The track file.
     */
    public Track(String artist, String title, String filename)
    {
        setDetails(artist, title, filename);
    }

    /**
     * Constructor for objects of class Track.
     * It is assumed that the file name cannot be
     * decoded to extract artist and title details.
     * @param filename The track file.
     */
    public Track(String filename)
    {
        setDetails("unknown", "unknown", filename);
    }

    /**
     * Return the artist.
     * @return The artist.
     */
    public String getArtist()
    {
        return artist;
```

```
    }

    /**
     * Return the title.
     * @return The title.
     */
    public String getTitle()
    {
        return title;
    }

    /**
     * Return the file name.
     * @return The file name.
     */
    public String getFilename()
    {
        return filename;
    }

    /**
     * Return details of the track: artist, title and file name.
     * @return The track's details.
     */
    public String getDetails()
    {
        return artist + ": " + title + "  (file: " + filename + ")";
    }

    /**
     * Set details of the track.
     * @param artist The track's artist.
     * @param title The track's title.
     * @param filename The track file.
     */
    private void setDetails(String artist, String title, String filename)
    {
        this.artist = artist;
        this.title = title;
        this.filename = filename;
    }

}
```

[ ]:
```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.InputStream;
```

```
import java.io.IOException;
import javazoom.jl.decoder.JavaLayerException;
import javazoom.jl.player.AudioDevice;
import javazoom.jl.player.FactoryRegistry;
import javazoom.jl.player.advanced.AdvancedPlayer;

/**
 * Provide basic playing of MP3 files via the javazoom library.
 * See http://www.javazoom.net/
 *
 * @author David J. Barnes and Michael Kölling.
 * @version 2016.02.29
 */
public class MusicPlayer
{
    // The current player. It might be null.
    private AdvancedPlayer player;

    /**
     * Constructor for objects of class MusicFilePlayer
     */
    public MusicPlayer()
    {
        player = null;
    }

    /**
     * Play a part of the given file.
     * The method returns once it has finished playing.
     * @param filename The file to be played.
     */
    public void playSample(String filename)
    {
        try {
            setupPlayer(filename);
            player.play(500);
        }
        catch(JavaLayerException e) {
            reportProblem(filename);
        }
        finally {
            killPlayer();
        }
    }

    /**
     * Start playing the given audio file.
```

```
     * The method returns once the playing has been started.
     * @param filename The file to be played.
     */
    public void startPlaying(final String filename)
    {
        try {
            setupPlayer(filename);
            Thread playerThread = new Thread() {
                public void run()
                {
                    try {
                        player.play(5000);
                    }
                    catch(JavaLayerException e) {
                        reportProblem(filename);
                    }
                    finally {
                        killPlayer();
                    }
                }
            };
            playerThread.start();
        }
        catch (Exception ex) {
            reportProblem(filename);
        }
    }

    public void stop()
    {
        killPlayer();
    }

    /**
     * Set up the player ready to play the given file.
     * @param filename The name of the file to play.
     */
    private void setupPlayer(String filename)
    {
        try {
            InputStream is = getInputStream(filename);
            player = new AdvancedPlayer(is, createAudioDevice());
        }
        catch (IOException e) {
            reportProblem(filename);
            killPlayer();
        }
```

```
        catch(JavaLayerException e) {
            reportProblem(filename);
            killPlayer();
        }
    }

    /**
     * Return an InputStream for the given file.
     * @param filename The file to be opened.
     * @throws IOException If the file cannot be opened.
     * @return An input stream for the file.
     */
    private InputStream getInputStream(String filename)
        throws IOException
    {
        return new BufferedInputStream(
                    new FileInputStream(filename));
    }

    /**
     * Create an audio device.
     * @throws JavaLayerException if the device cannot be created.
     * @return An audio device.
     */
    private AudioDevice createAudioDevice()
        throws JavaLayerException
    {
        return FactoryRegistry.systemRegistry().createAudioDevice();
    }

    /**
     * Terminate the player, if there is one.
     */
    private void killPlayer()
    {
        synchronized(this) {
            if(player != null) {
                player.stop();
                player = null;
            }
        }
    }

    /**
     * Report a problem playing the given file.
     * @param filename The file being played.
     */
```

```
    private void reportProblem(String filename)
    {
        System.out.println("There was a problem playing: " + filename);
    }

}
```

[ ]:
```
import java.io.File;
import java.io.FilenameFilter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.stream.Collectors;

/**
 * A helper class for our music application. This class can read files from the␣
 ↪file system
 * from a given folder with a specified suffix. It will interpret the file name␣
 ↪as artist/
 * track title information.
 *
 * It is expected that file names of music tracks follow a standard format of␣
 ↪artist name
 * and track name, separated by a dash. For example: TheBeatles-HereComesTheSun.
 ↪mp3
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class TrackReader
{
    /**
     * Create the track reader, ready to read tracks from the music library␣
 ↪folder.
     */
    public TrackReader()
    {
        // Nothing to do here.
    }

    /**
     * Read music files from the given library folder
     * with the given suffix.
     * @param folder The folder to look for files.
     * @param suffix The suffix of the audio type.
     */
    public ArrayList<Track> readTracks(String folder, String suffix)
```

```
    {
        File audioFolder = new File(folder);
        File[] audioFiles = audioFolder.listFiles((dir, name) ->
                    name.toLowerCase().endsWith(suffix));

        // Put all the matching files into the organizer.
        ArrayList<Track> tracks =
            Arrays.stream(audioFiles).
                   map(file -> decodeDetails(file)).
                   collect(Collectors.toCollection(ArrayList::new));
        return tracks;
    }

    /**
     * Try to decode details of the artist and the title
     * from the file name.
     * It is assumed that the details are in the form:
     *     artist-title.mp3
     * @param file The track file.
     * @return A Track containing the details.
     */
    private Track decodeDetails(File file)
    {
        // The information needed.
        String artist = "unknown";
        String title = "unknown";
        String filename = file.getPath();

        // Look for artist and title in the name of the file.
        String details = file.getName();
        String[] parts = details.split("-");

        if(parts.length == 2) {
            artist = parts[0];
            String titlePart = parts[1];
            // Remove a file-type suffix.
            parts = titlePart.split("\\.");
            if(parts.length >= 1) {
                title = parts[0];
            }
            else {
                title = titlePart;
            }
        }
        return new Track(artist, title, filename);
    }
}
```

[ ]:
```
import java.util.ArrayList;

/**
 * A class to hold details of audio tracks.
 * Individual tracks may be played.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class MusicOrganizer
{
    // An ArrayList for storing music tracks.
    private ArrayList<Track> tracks;
    // A player for the music tracks.
    private MusicPlayer player;
    // A reader that can read music files and load them as tracks.
    private TrackReader reader;

    /**
     * Create a MusicOrganizer
     */
    public MusicOrganizer()
    {
        tracks = new ArrayList<>();
        player = new MusicPlayer();
        reader = new TrackReader();
        readLibrary("../audio");
        System.out.println("Music library loaded. " + getNumberOfTracks() + "␣
↪tracks.");
        System.out.println();
    }

    /**
     * Add a track file to the collection.
     * @param filename The file name of the track to be added.
     */
    public void addFile(String filename)
    {
        tracks.add(new Track(filename));
    }

    /**
     * Add a track to the collection.
     * @param track The track to be added.
     */
    public void addTrack(Track track)
    {
```

```
        tracks.add(track);
    }

    /**
     * Play a track in the collection.
     * @param index The index of the track to be played.
     */
    public void playTrack(int index)
    {
        if(indexValid(index)) {
            Track track = tracks.get(index);
            player.startPlaying(track.getFilename());
            System.out.println("Now playing: " + track.getArtist() + " - " +␣
↪track.getTitle());
        }
    }

    /**
     * Return the number of tracks in the collection.
     * @return The number of tracks in the collection.
     */
    public int getNumberOfTracks()
    {
        return tracks.size();
    }

    /**
     * List a track from the collection.
     * @param index The index of the track to be listed.
     */
    public void listTrack(int index)
    {
        System.out.print("Track " + index + ": ");
        Track track = tracks.get(index);
        System.out.println(track.getDetails());
    }

    /**
     * Show a list of all the tracks in the collection.
     */
    public void listAllTracks()
    {
        System.out.println("Track listing: ");

        for(Track track : tracks) {
            System.out.println(track.getDetails());
        }
```

```
        System.out.println();
    }

    /**
     * List all tracks by the given artist.
     * @param artist The artist's name.
     */
    public void listByArtist(String artist)
    {
        for(Track track : tracks) {
            if(track.getArtist().contains(artist)) {
                System.out.println(track.getDetails());
            }
        }
    }

    /**
     * Remove a track from the collection.
     * @param index The index of the track to be removed.
     */
    public void removeTrack(int index)
    {
        if(indexValid(index)) {
            tracks.remove(index);
        }
    }

    /**
     * Play the first track in the collection, if there is one.
     */
    public void playFirst()
    {
        if(tracks.size() > 0) {
            player.startPlaying(tracks.get(0).getFilename());
        }
    }

    /**
     * Stop the player.
     */
    public void stopPlaying()
    {
        player.stop();
    }

    /**
     * Determine whether the given index is valid for the collection.
```

```
     * Print an error message if it is not.
     * @param index The index to be checked.
     * @return true if the index is valid, false otherwise.
     */
    private boolean indexValid(int index)
    {
        // The return value.
        // Set according to whether the index is valid or not.
        boolean valid;

        if(index < 0) {
            System.out.println("Index cannot be negative: " + index);
            valid = false;
        }
        else if(index >= tracks.size()) {
            System.out.println("Index is too large: " + index);
            valid = false;
        }
        else {
            valid = true;
        }
        return valid;
    }

    private void readLibrary(String folderName)
    {
        ArrayList<Track> tempTracks = reader.readTracks(folderName, ".mp3");

        // Put all thetracks into the organizer.
        for(Track track : tempTracks) {
            addTrack(track);
        }
    }
}
```

1.4 Iterator Objects

All Java collections have an `iterator()` method, which returns an `Iterator` object. This provides sequential access to the whole collection. This has methods:

- boolean `hasNext()`
  - a boolean that indicates if there is another object in the collection
  - goes false after the last item in the collection has been visited
- `next()`, which has the same type as the original object
  - next object in the list

- void `remove()`
  - removes the element currently pointed to by `iterator()`

1.5 Index versus `Iterator`

We can traverse a collection by:

- for each loop - processes every element
- while loop - if we want to stop, or don't have a collection
- `Iterator`
  - use if you might stop partway
  - used if indexed access is not efficient or sensible
  - **used to remove things from a collection**

1.5.1 Removing from a collection

To remove an item where you know what the item is but not necessarily where it is in the collection:

- create the iterator: `Iterator<Track> it = tracks.iterator();`
- iterate through the list until you get the object to remove
- call `it.remove()`

This almost sucks more than doing it in a C array. WTF.

1.5.2 `null`

In Java, `null` is a special value for references, not entirely unlke the `NULL` pointer in C.

`null` is the default reference for an object that hasn't been created yet, hasn't been attached, or for whatever reason just doesn't point anywhere yet.

We can test if a reference holds `null` by `if ( something == null )`.

1.5.3 Anonymous objects

We can create some objects and pass them off *without naming them*: `Lot furtherlot = new Lot(); lots.add(furtherlot);` can be written more concisely as `lots.add( new Lot() );`. This is called an *anonymous object*.

1.5.4 Chaining method calls

If a method returns an object, we can call one of that object's method directly without putting it anywhere. If `lot` has a method `getHighestBid()` which returns a `Bid` object, and `Bid` has a method `getBidder()`, we can just chain together `lot.getHighestBid().getBidder()` to get `Bidder`'s attribute.

[ ]: